1、 响应式布局如何实现

响应式布局可以让网站同时适配不同分辨率和不同的手机端，让客户有更好的体验。

方案一：百分比布局

利用对属性设置百分比来适配不同屏幕，注意这里的百分比是相对于父元素；能够设置的属性有 width、height、padding、margin，其他属性比如 border、font-size 不能用百分比设置的，先看一个简单例子：


```
.wrapper .left {
    width: 50%;
}

.wrapper .right {
    width: 50%;
}

.tabBox {
    margin-top: 20px;
}

.tabBox li {
    width: 25%;
    display: inline-block;
    text-align: center;
    border-right: 1px dashed #333;
    box-sizing: border-box;
}

.tabBox li.last {
    border-right: none;
}
</style>
ead>

dy>
<div class="wrapper"><img src="images/left.jpg" alt="" class="left"><img src="images/right.jp
right"></div>
<ul class="tabBox">
    <li>栏目1</li><li>栏目2</li><li>栏目3</li><li class="last">栏目4</li>
</ul>
```


顶部是利用设置图片 width: 50%来适应不同的分辨率，由于原始图片高度不同，所以第一张图片顶部会有空白，这种情况最好两张图片宽高保持一致，如果使用强制高度统一，会导致图片变形；

注意：当屏幕大于图片的宽度时，会进行拉伸；解决拉伸方法就是改为 max-width: 50%，但当屏幕大于图片的宽度时，两边会有空白。

栏目是利用设置单栏目 width: 25%来适应不同的分辨率。

由于没办法对 font-size 进行百分比设置，所以用的最多就是对图片和大块布局进行百分比设置。

方案二：使用媒体查询 (CSS3 @media 查询)

利用媒体查询设置不同分辨率下的 css 样式，来适配不同屏幕，先看一个简单例子：


```
body {
    background-color: grey;
}

@media screen and (min-width:1200px) {
    body {
        background-color: pink;
    }
}

@media screen and (min-width:1200px) and (max-width:1200px) {
    body {
        background-color: blue;
    }
}

@media screen and (max-width:700px) {
    body {
        background-color: orange;
    }
}
```

三个不同分辨率下显示对应的背景色。

媒体查询相对于百分比布局，可以对布局进行更细致的调整，但需要在每个分辨率下面都写一套 css 样式；分辨率拆分可视项目具体情况而定。

注意：IE6、7、8 不支持媒体查询。

方案三.rem 响应式布局

当前页面中元素的 rem 单位的样式值都是针对于 html 元素的 font-size 的值进行动态计算的，所以有两种方法可以达到适配不同屏幕：

第一种利用媒体查询，在不同分辨率下给 html 的 font-size 赋值。

第二种利用 js 动态计算赋值，详细代码如下图：


```
// 动态计算屏幕的宽度，从而得到网页的fontSize大小
(function(doc, win) {
    var docEl = doc.documentElement,
        resizeEvt = 'orientationchange' in window ? 'orientationchange' : 'resize',
        recalc = function() {
            var clientWidth = docEl.clientWidth;
            if (!clientWidth) return;
            if (clientWidth > 750) clientWidth = 750; //这里限制最大的宽度尺寸，从而实现PC端的两边留白等
            docEl.style.fontSize = 10 * (clientWidth / 320) + 'px';
        };

    if (!doc.addEventListener) return;
    win.addEventListener(resizeEvt, recalc, false);
    doc.addEventListener('DOMContentLoaded', recalc, false);
})(document, window);
```

缺点就是打开页面时候，元素大小会有一个变化过程。

方案四.vw 响应式布局

根据 PSD 文件宽度或高度作为标准，元素单位 px 转换为 vw 或 vh，比如 font-size: 12px，PSD 文件宽度 375，转换公式 12 * 100 / 375，则样式改为 font-size: 3.2vw，下面是我经常使用的工具，有利于提高转换效率。

现阶段手机端用的最多就是这个方法，能保持不同屏幕下元素显示效果一致，也不用写多套样式。

方案五.flex 弹性布局

利用 flex 属性来适配不同屏幕，下图利用简单的属性实现栏目响应式

2、 rem 布局原理

rem：相对于根元素(即 html 元素)font-size 计算值的倍数。

通俗的说，**1rem = html 的 font-size 值**

这段代码。a 标签的 font-size 值为 0.5rem，实际就是 100px*0.5=50px。

html{font-size:100px;}　　a{font-size:.5rem;}

如何使用 rem 进行布局？

1. 标签的 rem 单位的值怎么计算

通过使用 rem + js 改变 html 标签的 font-size（整体缩放）实现兼容性更高的页面下面来举个例子，

当我们拿到的设计图是 750px 的时候，窗口宽度 750px，html 的 font-size 的大小为 100px；

也就是说 1rem = 100px；所以标题的 font-size 的大小为 26/100=.26rem；

2. 如何实现兼容各种屏幕大小的设备

使用到 javascript 来动态改变 html 标签 font-size 的大小，其他的 rem 单位的数值就会被浏览动态计算转为 px 单位，从而达到和设计图高度的相似。

当屏幕 750px 的时候，html 的 font-size 值是 100px；窗口大小变化的时候，可以通过 js 获取到窗口大小。

这时候获取到一个比例 750：100=获取到的屏幕大小：html 标签的 px 单位的值

以下 js 代码，用于实现根据获取到的屏幕大小，动态修改 html 标签的 px 单位的值

```
;(function(designWidth, maxWidth) {
    var doc = document,
        win = window;
    var docEl = doc.documentElement;
    var tid;
    var rootItem,rootStyle;

    function refreshRem() {
        var width = docEl.getBoundingClientRect().width;
        if (!maxWidth) {
            maxWidth = 640;
        };
        if (width > maxWidth) {
            width = maxWidth;
        }
        var rem = width * 100 / designWidth;
        //兼容UC开始
        rootStyle="html{font-size:"+rem+'px !important}';
        rootItem = document.getElementById('rootsize') ||
document.createElement("style");
        if(!document.getElementById('rootsize')){
        document.getElementsByTagName("head")[0].appendChild(rootItem);
        rootItem.id='rootsize';
        }
        if(rootItem.styleSheet){
        rootItem.styleSheet.disabled||(rootItem.styleSheet.cssText=rootStyle)
        }else{
        try{rootItem.innerHTML=rootStyle}catch(f){rootItem.innerText=rootStyle}
        }
        //兼容UC结束
        docEl.style.fontSize = rem + "px";
    };
    refreshRem();

    win.addEventListener("resize", function() {
        clearTimeout(tid); //防止执行两次
        tid = setTimeout(refreshRem, 300);
    }, false);

    win.addEventListener("pageshow", function(e) {
        if (e.persisted) { // 浏览器后退的时候重新计算
            clearTimeout(tid);
            tid = setTimeout(refreshRem, 300);
        }
    }, false);

    if (doc.readyState === "complete") {
        doc.body.style.fontSize = "16px";
    } else {
        doc.addEventListener("DOMContentLoaded", function(e) {
            doc.body.style.fontSize = "16px";
        }, false);
    }
})(750, 750);
```

3、 三种方式实现一个 div 水平和垂直方向的居中

1.

```
<style>
  #box{
    height: 200px;
    width: 200px;
    position: absolute;
    left: 50%;
    top: 50%;
    margin-left: -100px;
    margin-top: -100px;
    background: red;
  }
</style>
<body>

  <div id="box"></div>
</body>
```

2.

```
<style>
  #img1{
      position: absolute;
      left: 0;
      top: 0;
      bottom: 0;
      right: 0;
      margin:auto;
  }
</style>
<body>
  <img src="./image/image_1.png" alt="" id="img1" >

</body>
```

3.

```
    <style>
        *{
            margin: 0;
            padding: 0;
            list-style-type: none;
        }
        html,body{
            height:100%;
            overflow:hidden;
        }
        #box1{
            width:100%;
            height:100%;
            background:skyblue;
            display:flex;
        }
        #img1{
            margin:auto;
        }
    </style>
</head>
<body>
    <div id="box1">
        <img id="img1" src="./image/image_1.png" alt="">
    </div>
```

4、 数据类型判断

1. typeof　typeof 对于基本数据类型判断是没有问题的，但是遇到引用数据类型（如：Array）是不起作用
2. instanceof　判断 new 关键字创建的引用数据类型

   不考虑 null 和 undefined（这两个比较特殊）以对象字面量创建的基本数据类型

3. constructor constructor 似乎完全可以应对基本数据类型和引用数据类型 但如果声明了一个构造函数，并且把他的原型指向了 Array 的原型，所以这种情况下，constructor 也显得力不从心
4. Object.prototype.toString.call()　完美的解决方案

5、 Arguments

arguments 是一个类似数组的对象, 对应于传递给函数的参数。

1. 描述

arguments 对象是所有函数中可用的局部变量。你可以使用 arguments 对象在函数中引用函数的参数。

2. 属性

- arguments.callee 指向当前执行的函数。
- arguments.caller **指向调用当前函数的函数。
- arguments.length 指向传递给当前函数的参数数量。

3. 例子

定义一个连接几个字符串的函数

```
function myConcat(separator) {
    var args = Array.prototype.slice.call(arguments, 1);
    return args.join(separator);
}
```

剩余参数, 默认参数 和 解构赋值参数

```
function foo(...args) {
    return args;
}
foo(1, 2, 3);   // [1,2,3]
```

等

6、 原型和原型链

1、原型的概念

JavaScript 的所有对象中都包含了一个 [proto] 内部属性，这个属性所对应的就是自身的原型

JavaScript 的函数对象，除了原型 [proto] 之外，还有 prototype 属性，当函数对象作为构造函数创建实例时，该 prototype 属性值将被作为实例对象的

原型 [proto]

2、原型链的概念

当一个对象调用自身不存在的属性/方法时，就会去自己 [proto] 关联的前辈 prototype 对象上去找，如果没找到，就会去该 prototype 原型 [proto] 关联的前辈 prototype 去找。依次类推，直到找到属性/方法或 undefined 为止。从而形成了所谓的“原型链”。

3、总结

JavaScript 中的对象，都有一个内置属性[Prototype]，指向这个对象的原型对象。当查找一个属性或方法时，如果在当前对象中找不到，会继续在当前对象的原型对象中查找；如果原型对象中依然没有找到，会继续在原型对象的原型中查找（原型也是对象，也有它自己的原型）；直到找到为止，或者查找到最顶层的原型对象中也没有找到，就结束查找，返回 undefined。这个查找过程是一个链式的查找，每个对象都有一个到它自身原型对象的链接，这些链接组建的整个链条就是原型链。拥有相同原型的多个对象，他们的共同特征正是通过这种查找模式体现出来的。

在上面的查找过程，我们提到了最顶层的原型对象，这个对象就是 Object.prototype，这个对象中保存了最常用的方法，如 toString、valueOf、hasOwnProperty 等，因此我们才能在任何对象中使用这些方法。

7、 闭包

闭包:

定义 当一个函数的返回值是另外一个函数,而返回的那个函数如果调用了其父函数的内部变量,且返回的那个函数在外部被执行,就产生了闭包.

闭包是一个环境,具体指的就是外部函数--高阶函数 closure

闭包的三个特性

1:函数套函数

2:内部函数可以直接访问外部函数的内部变量或参数

3:变量或参数不会被垃圾回收机制回收 GC

闭包的优点:

1:变量长期驻扎在内存中

2:避免全局变量的污染

3:私有成员的存在

闭包的缺点

常驻内存　增大内存的使用量　使用不当会造成内存的泄露.

闭包的两种写法:

1：

```
function a () {
    var num=1;
    function b () {
        alert(num)

    }
    return b;//把函数 b 返回给函数 a;
}
alert(a())//弹出函数 a, 值是函数 b;
```

2：

```
function a () {
    var num=1;
    return function b () {//把函数 b 返回给函数 a;
        alert(num=num+2)
    }
}
alert(a())//弹出函数 a, 值是函数 b;
```

调用方式：

```
//1:直接调用
a()()//内部函数的执行

//2：通过赋值在调用
var f = a();
f()
```

8、 js 继承

ES5 有 6 种方式可以实现继承，分别为：

1. 原型链继承

原型链继承的基本思想是利用原型让一个引用类型继承另一个引用类型的属性和方法。

```
function SuperType() {
    this.name = 'Yvette';
    this.colors = ['pink', 'blue', 'green'];
}
SuperType.prototype.getName = function () {
    return this.name;
}
function SubType() {
    this.age = 22;
}
SubType.prototype = new SuperType();
SubType.prototype.getAge = function() {
    return this.age;
}
SubType.prototype.constructor = SubType;
let instance1 = new SubType();
instance1.colors.push('yellow');
console.log(instance1.getName()); //'Yvette'
console.log(instance1.colors);//[ 'pink', 'blue', 'green', 'yellow' ]

let instance2 = new SubType();
console.log(instance2.colors);//[ 'pink', 'blue', 'green', 'yellow' ]
```

缺点：

1 通过原型来实现继承时，原型会变成另一个类型的实例，原先的实例属性变成了现在的原型属性，该原型的引用类型属性会被所有的实例共享。

2 在创建子类型的实例时，没有办法在不影响所有对象实例的情况下给超类型的构造函数中传递参数。

2. 借用构造函数

借用构造函数的技术，其基本思想为:

在子类型的构造函数中调用超类型构造函数。

```
function SuperType(name) {
    this.name = name;
    this.colors = ['pink', 'blue', 'green'];
}
function SubType(name) {
    SuperType.call(this, name);
}
let instance1 = new SubType('Yvette');
instance1.colors.push('yellow');
console.log(instance1.colors);//['pink', 'blue', 'green', yellow]

let instance2 = new SubType('Jack');
console.log(instance2.colors); //['pink', 'blue', 'green']
```

优点:

1 可以向超类传递参数

2 解决了原型中包含引用类型值被所有实例共享的问题

缺点:

- 方法都在构造函数中定义，函数复用无从谈起，另外超类型原型中定义的方法对于子类型而言都是不可见的。

3. 组合继承(原型链 + 借用构造函数)

组合继承指的是将原型链和借用构造函数技术组合到一块，从而发挥二者之长的一种继承模式。基本思路：

使用原型链实现对原型属性和方法的继承，通过借用构造函数来实现对实例属性的继承，既通过在原型上定义方法来实现了函数复用，又保证了每个实例都有自己的属性。

```
function SuperType(name) {
    this.name = name;
    this.colors = ['pink', 'blue', 'green'];
}
SuperType.prototype.sayName = function () {
    console.log(this.name);
}
function SuberType(name, age) {
    SuperType.call(this, name);
    this.age = age;
}
SuberType.prototype = new SuperType();
SuberType.prototype.constructor = SuberType;
SuberType.prototype.sayAge = function () {
    console.log(this.age);
}
let instance1 = new SuberType('Yvette', 20);
instance1.colors.push('yellow');
console.log(instance1.colors); //[ 'pink', 'blue', 'green', 'yellow' ]
instance1.sayName(); //Yvette

let instance2 = new SuberType('Jack', 22);
console.log(instance2.colors); //[ 'pink', 'blue', 'green' ]
instance2.sayName();//Jack
```

缺点:

- 无论什么情况下，都会调用两次超类型构造函数：一次是在创建子类型原型的时候，另一次是在子类型构造函数内部。

优点:

- 可以向超类传递参数
- 每个实例都有自己的属性
- 实现了函数复用

4. 原型式继承

原型继承的基本思想：

借助原型可以基于已有的对象创建新对象，同时还不必因此创建自定义类型。

```
function object(o) {
    function F() { }
    F.prototype = o;
    return new F();
}
```

在 object() 函数内部，先穿甲一个临时性的构造函数，然后将传入的对象作为这个构造函数的原型，最后返回了这个临时类型的一个新实例，从本质上讲，object() 对传入的对象执行了一次浅拷贝。

ECMAScript5 通过新增 Object.create()方法规范了原型式继承。这个方法接收两个参数：一个用作新对象原型的对象和（可选的）一个为新对象定义额外属性的对象(可以覆盖原型对象上的同名属性)，在传入一个参数的情况下，Object.create() 和 object() 方法的行为相同。

```
var person = {
    name: 'Yvette',
    hobbies: ['reading', 'photography']
}
var person1 = Object.create(person);
person1.name = 'Jack';
person1.hobbies.push('coding');
var person2 = Object.create(person);
person2.name = 'Echo';
person2.hobbies.push('running');
console.log(person.hobbies);//[ 'reading', 'photography', 'coding', 'running' ]
console.log(person1.hobbies);//[ 'reading', 'photography', 'coding', 'running' ]
```

在没有必要创建构造函数，仅让一个对象与另一个对象保持相似的情况下，原型式继承是可以胜任的。

缺点:

同原型链实现继承一样，包含引用类型值的属性会被所有实例共享。

5. 寄生式继承

寄生式继承是与原型式继承紧密相关的一种思路。寄生式继承的思路与寄生构造函数和工厂模式类似，即创建一个仅用于封装继承过程的函数，该函数在内部已某种方式来增强对象，最后再像真地是它做了所有工作一样返回对象。

```
function createAnother(original) {
    var clone = object(original);//通过调用函数创建一个新对象
    clone.sayHi = function () {//以某种方式增强这个对象
        console.log('hi');
    };
    return clone;//返回这个对象
}
var person = {
    name: 'Yvette',
    hobbies: ['reading', 'photography']
};

var person2 = createAnother(person);
person2.sayHi(); //hi
```

基于 person 返回了一个新对象 ——— person2，新对象不仅具有 person 的所有属性和方法，而且还有自己的 sayHi() 方法。在考虑对象而不是自定义类型和构造函数的情况下，寄生式继承也是一种有用的模式。

缺点：

- 使用寄生式继承来为对象添加函数，会由于不能做到函数复用而效率低下。
- 同原型链实现继承一样，包含引用类型值的属性会被所有实例共享。

6. 寄生组合式继承

所谓寄生组合式继承，即通过借用构造函数来继承属性，通过原型链的混成形式来继承方法，基本思路：

不必为了指定子类型的原型而调用超类型的构造函数，我们需要的仅是超类型原型的一个副本，本质上就是使用寄生式继承来继承超类型的原型，然后再将结果指定给子类型的原型。寄生组合式继承的基本模式如下所示：

```
function inheritPrototype(subType, superType) {
    var prototype = object(superType.prototype); //创建对象
    prototype.constructor = subType;//增强对象
    subType.prototype = prototype;//指定对象
}
```

- 第一步：创建超类型原型的一个副本
- 第二步：为创建的副本添加 constructor 属性
- 第三步：将新创建的对象赋值给子类型的原型

至此，我们就可以通过调用 inheritPrototype 来替换为子类型原型赋值的语句：

```
function SuperType(name) {
    this.name = name;
    this.colors = ['pink', 'blue', 'green'];
}
//...code
function SuberType(name, age) {
    SuperType.call(this, name);
    this.age = age;
}
SuberType.prototype = new SuperType();
inheritPrototype(SuberType, SuperType);
//...code
```

优点：

只调用了一次超类构造函数，效率更高。避免在 SuberType.prototype

上面创建不必要的、多余的属性，与其同时，原型链还能保持不变。因此寄生组合继承是引用类型最理性的继承范式。

ES6 实现继承

```
class VipUser extends User{
    constructor(name,pass,level){
      super(name,pass)//指针 指向父类构造器  区别于this
      this.level = level;
    }
    showLevel(){
        console.log(this.level)
    }
}

var u1 = new VipUser("tom","123456",3)
u1.showName();
u1.showPass();
u1.showLevel()
console.log(typeof VipUser)
```

9、 什么是深拷贝，浅拷贝，如何实现

深拷贝和浅拷贝是针对复杂数据类型来说的，浅拷贝只拷贝一层，而深拷贝是层层拷贝。

深拷贝

深拷贝复制变量值，对于非基本类型的变量，则递归至基本类型变量后，再复制。 深拷贝后的对象与原来的对象是完全隔离的，互不影响，对一个对象的修改并不会影响另一个对象。

浅拷贝

浅拷贝是会将对象的每个属性进行依次复制，但是当对象的属性值是引用类型时，实质复制的是其引用，当引用指向的值改变时也会跟着变化。

可以使用 for in 、 Object.assign 、 扩展运算符 ... 、 Array.prototype.slice()、Array.prototype.concat() 、递归等递归函数实现深拷贝

```
function deepClone(obj, hash = new WeakMap()) { //递归拷贝
    if (obj instanceof RegExp) return new RegExp(obj);
    if (obj instanceof Date) return new Date(obj);
    if (obj === null || typeof obj !== 'object') {
        //如果不是复杂数据类型，直接返回
        return obj;
    }
    if (hash.has(obj)) {
        return hash.get(obj);
    }
    /**
     * 如果obj是数组，那么 obj.constructor 是 [Function: Array]
     * 如果obj是对象，那么 obj.constructor 是 [Function: Object]
     */
    let t = new obj.constructor();
    hash.set(obj, t);
    for (let key in obj) {
        //递归
        if (obj.hasOwnProperty(key)) {//是否是自身的属性
            t[key] = deepClone(obj[key], hash);
        }
    }
    return t;
}
```

10、事件冒泡，事件捕获

什么是事件？

事件是文档和浏览器窗口中发生的特定的交互瞬间。 事件是 javascript 应用跳动的心脏，也是把所有东西黏在一起的胶水，当我们与浏览器中 web 页面进行某些类型的交互时，事件就发生了。

事件可能是用户在某些内容上的点击，鼠标经过某个特定元素或按下键盘上的某些按键，事件还可能是 web 浏览器中发生的事情，比如说某个 web 页面加载完成，或者是用户滚动窗口或改变窗口大小。

什么是事件流？

事件流描述的是从页面中接受事件的顺序，但有意思的是，微软（IE）和网景（Netscape）开发团队居然提出了两个截然相反的事件流概念，IE 的事件流是事件冒泡流(event bubbling)，而 Netscape 的事件流是事件捕获流(event capturing)。

事件冒泡和事件捕获的概念：

事件冒泡和事件捕获是描述事件触发事件时序问题的术语，事件捕获指的是从 document 到触发事件的那个节点，也就是说自上而下的去触发事件，相反的，事件冒泡是自下而上的去触发事件，绑定事件方法的第三个参数，就是控制事件触发顺序是否为事件捕获，true 为事件捕获，false 为事件冒泡，jQuery 的 e.stopPropagation 会阻止冒泡，意思就是到我为止，我的爹和祖宗的事件就不要触发了。

第一种：事件冒泡

IE 提出的事件流叫做事件冒泡，即事件开始时由最具体的元素接收，然后逐级向上传播到较为不具体的节点

```
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body onclick="bodyClick()">

    <div onclick="divClick()">
        <button onclick="btn()">
            <p onclick="p()">点击冒泡</p>
        </button>
    </div>
    <script>

       function p(){
          console.log('p标签被点击')
       }
        function btn(){
            console.log("button被点击")
        }
         function divClick(event){
             console.log('div被点击');
         }
        function bodyClick(){
            console.log('body被点击')
        }

    </script>
```

执行顺序： p=>button=>div=>body

正如上面我们所说的，它会从一个最具体的的元素接收，然后逐级向上传播，p=>button=>div=>body.....事件冒泡可以形象地比喻为把一颗石头投入水中，泡泡会一直从水底冒出水面，也就是说从小到大开始传播。

第二种：事件捕获

网景公司提出的事件流叫事件捕获流。

事件捕获流的思想是不太具体的 DOM 节点应该更早接收到事件，而最具体的节点应该最后接收到事件，针对上面同样的例子，点击按钮，那么此时 click 事件会按照这样传播：（下面我们就借用 addEventListener 的第三个参数来模拟事件捕获流），也就是上面的例子就会倒过来。

```
<body>

<div>
    <button>
        <p>点击捕获</p>
    </button>
</div>
<script>
    var oP=document.querySelector('p');
    var oB=document.querySelector('button');
    var oD=document.querySelector('div');
    var oBody=document.querySelector('body');

    oP.addEventListener('click',function(){
        console.log('p标签被点击')
    },true);

    oB.addEventListener('click',function(){
        console.log("button被点击")
    },true);

    oD.addEventListener('click',  function(){
        console.log('div被点击')
    },true);

    oBody.addEventListener('click',function(){
        console.log('body被点击')
    },true);
```

正如我们看到的，和冒泡流万全相反，从最不具体的元素接收到最具体的元素接收事件 body=>div=>button=>p

11、 h5 和 css3 的新特性

h5 每个人有每个人的理解,我的理解呢！我的理解是 h5 呢并不是新增一些标签和样式更多的是里面新增的一些功能例如重力感应，他可以让我们感知当前手机的状态，可以帮助我们完成手机摇一摇，监听当前我们步数，还有开启 3d 模式让我们的 2d 空间变成一个 3d 模式,而且 h5 中为了挺高页面性能，页面元素的变大，不在是元素本身的大小变化，而是一种视觉上的效果，从而减少了 dom 操作，防止了页面的重绘，当然 h5 中不单单是这些还有 webgl 游戏引擎 canvas、svg 完成图表以及一些小游戏的开发例如大转盘，数钱游戏，jd 妈妈再打我一次等等！

12、 http/https/http2.0

HTTP 是一种协议，也是一种请求，只要是你调用接口的，或是你发送请求的时候都是发送一种 HTTP 请求，然后我们常用的那种 HTTP 请求的话，比如说 GET 啊，POST 啊，这种这种请求的话，其实，只要是你发送一次请求，它是两个都是存在的，GET、POST 的是同时存在的。然后 GET 的话在 HTTP 的头上，POST 在那身体里面。的话，然后因为在头部的话，所以说它传输量比较小一点，一般来说是 32k，POST 在身体上比较大一点，大约是 2G,而且 post 的话能够传输所有的二进制文件和数据,GET 只能接受数据,GET 这种请求因为在地址栏上所

以它更不安全，post 这种因为在身体上所以它相对安全，GET 是有缓存，POST 是没有缓存,但是 GET 请求一直存在，主要原因还是 GET 这种请求的话,可以做分享和收藏，美丽说，蘑菇街都是通过分享和收藏火起来，并且发展起来，所以 GET 的作用不可小觑，这个就是我理解的 http 请求,而且 http 请求的话是一种无状态的请求所以，我们会用 session 或是 token 去记录它当前的状态，或是他当前的权限，比如只有你登录以后你才可以获取到他的数据，https 这种请求是对 http 请求的再次封装以后的结果,接下来我就说一说他与 http 请求的区别 https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用，这是第一点，第二点就是 http 这种请求是明文传输，https 则是具有安全性的 ssl 加密传输协议，第三点 http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443 最后一点就是 http 的连接很简单，是无状态的，HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全

13、 Axios 拦截做过哪些

Axios 拦截分为请求拦截和响应拦截，请求拦截就是在你请求的时候会进行触发！只要是你发送一个 axios 请求就会触发！所以我们主要用它做我们的 loading 加载和数据的权限验证,包括我们所有的数据预加载也可以实现，响应拦截主要是我们在 loading 加载，和做所有数据加载需要整体的结束，这个时候的结束就需要在数据马上发给前端的时候进行隐藏和结束，包括我们的请求头的设置，后端数据已经发送过来的时候，我们为了确保请求头的传递就必须在看看 header 里面是否有你需要的请求，如果有的话，再次进行设置！当然用 axios 拦截也可以配置公用地址，以及对于跨域问题解决，这个就是用 axios 拦截实现的功能。

14、 sessionStoragelocalStorage cookie 的区别

1．localStorage 生命周期是永久，这意味着除非用户显示在浏览器提供的 UI 上清除 localStorage 信息，否则这些信息将永远存在。存放数据大小为一般为 5MB,而且它仅在客户端（即浏览器）中保存，不参与和服务器的通信。

2．sessionStorage 仅在当前会话下有效，关闭页面或浏览器后被清除。存放数据大小为一般为 5MB,而且它仅在客户端（即浏览器）中保存，不参与和服务器的通信。源生接口可以接受，亦可再次封装来对 Object 和 Array 有更好的支持。

作用域不同

不同浏览器无法共享 localStorage 或 sessionStorage 中的信息。相同浏览器的不同页面间可以共享相同的 localStorage（页面属于相同域名和端口），但是不同页面或标签页间无法共享 sessionStorage 的信息。这里需要注意的是，页面及标

3. cookie 的优点：具有极高的扩展性和可用性

1.通过良好的编程，控制保存在 cookie 中的 session 对象的大小。

2.通过加密和安全传输技术，减少 cookie 被破解的可能性。

3.只有在 cookie 中存放不敏感的数据，即使被盗取也不会有很大的损失。

4.控制 cookie 的生命期，使之不会永远有效。这样的话偷盗者很可能拿到的就是一个过期的 cookie。</pre>

4. cookie 的缺点：

1.cookie 的长度和数量的限制。每个 domain 最多只能有 20 条 cookie，每个 cookie 长度不能超过 4KB。否则会被截掉。

2.安全性问题。如果 cookie 被人拦掉了，那个人就可以获取到所有 session 信息。加密的话也不起什么作用。

3.有些状态不可能保存在客户端。例如，为了防止重复提交表单，我们需要在服务端保存一个计数器。若吧计数器保存在客户端，则起不到什么作用。</pre>

localStorage、sessionStorage、Cookie 共同点：都是保存在浏览器端，且同源的。

15、 图片懒加载实现原理

一.什么是懒加载？

懒加载突出一个“懒”字，懒就是拖延迟的意思，所以“懒加载”说白了就是延迟加载，比如我们加载一个页面，这个页面很长很长，长到我们的浏览器可视区域装不下，那么懒加载就是优先加载可视区域的内容，其他部分等进入了可视区域在加载。

二.为什么要懒加载？

懒加载是一种网页性能优化的方式，它能极大的提升用户体验。就比如说图片，图片一直是影响网页性能的主要元凶，现在一张图片超过几兆已经是很经常的事了。如果每次进入页面就请求所有的图片资源，那么可能等图片加载

出来用户也早就走了。所以，我们需要懒加载，进入页面的时候，只请求可视区域的图片资源。

总结出来就两个点：

1.全部加载的话会影响用户体验

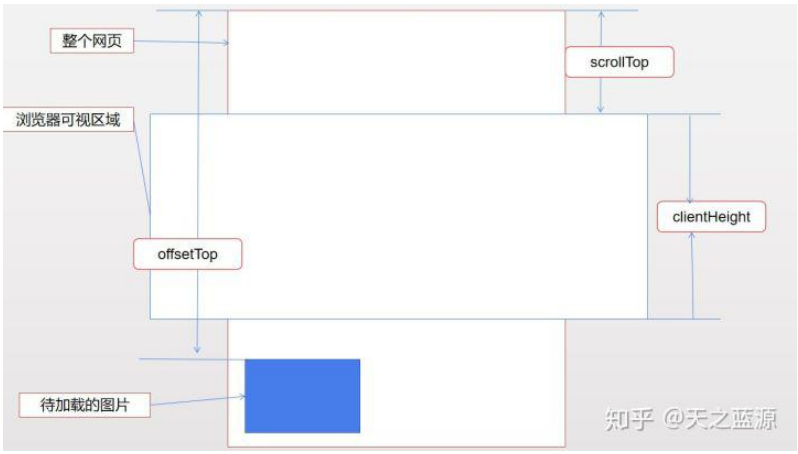
2.浪费用户的流量，有些用户并不像全部看完，全部加载会耗费大量流量。

三.懒加载的实现原理？

由于网页中占用资源较多的一般是图片，所以我们一般实施懒加载都是对图片资源而言的，所以这里的实现原理主要是针对图片。

大家都知道，一张图片就是一个<img>标签，而图片的来源主要是 src 属性。浏览器是否发起亲求就是根据是否有 src 属性决定的。

既然这样，那么我们就要对<img>标签的 src 属性下手了，在没进入可视区域的时候，我们先不给这个<img>标签赋 src 属性，这样岂不是浏览器就不会发送请求了。



16、 瀑布流原理

瀑布流又称瀑布流式布局，是比较流行的一种网站页面布局方式

视觉表现为参差不齐的多栏布局，最早采用此布局的是网站是 Pinterest，后逐渐在国内流行即多行等宽元素排列，后面的元素依次添加到其后，等宽不等高，根据图片原比例缩放直至宽度达到我们的要求，依次按照规则放入指定位置。

下面通过图解来分析一下瀑布流的算法。

当第一排排满足够多的等宽图片时（如下图情况），自然而然的考虑到之后

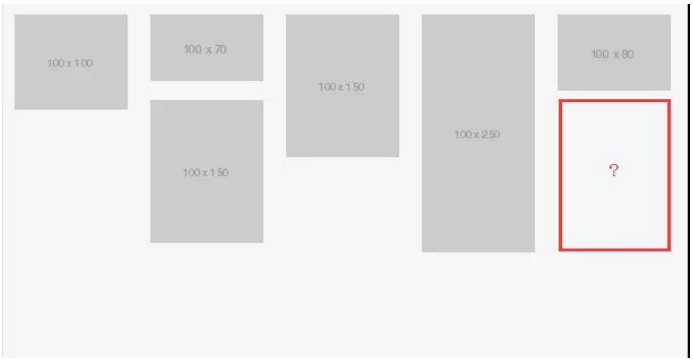
放置的图片会往下面排放。

那么第六张图片，放置在什么位置呢？是下图的位置么？

我们通过瀑布流算法实验得到，后面紧跟的第六张图片的位置应该是这个位置。

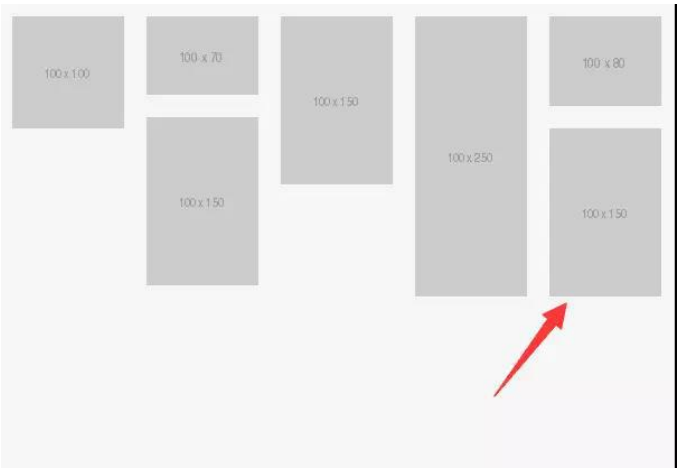
因为放置它之前，这一列的高度为所有列中最小，所以会放置在这个地方。

所以我们知道了，如果再继续放置下去，第七张图片应该是这个位置



通过瀑布流算法实验得出位置正确。

看懂这个图示应该就能理解了瀑布流的原理算法。



总结瀑布流布局原理：

瀑布流的实现原理就是通过比较每一列的高度，如果这一列高度低的话就在这一列进行添加，如果可视区距离和滚动距离想加的话比你当前页面的高度还要高的时候，页面就开再次加载多个，这个主要也是页面布局，如果你的布局实现的不好的话！也会出现问题，首先每一列的高度，都是需要自适应的，不能设置高通过每一块内容将每一列撑起来，还有一个问题就是，滑动底部，根据每个电脑的不同，所以他每个人,获取的高度是不同而且有的时候，页面的整体高度和页面可视区高度加上滚动高度差 1px 这个时候就需要我们提前加载，不然滑动到底部也加载不出来！

17、 Call、 apply 区别，原生实现 bind

Call、apply、bind 都是改变 this 指向的，其中 call 的话是通过第一个参数来改变 this 指向，函数在传参的时候一个一个进行传参，apply 也是通过第一个参数改变 this 指向，函数传参的时候通过数组或是一组的形式进行传参，而且他只能用于函数的调用时候，而 bind 呢不能用于函数的标准调用传参，只能是事件或是方法的后边进行改变 this 的指向

18、 解构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring)

解构赋值，左右结构必须一样，使用左边定义的值，快速的取出数据中对应的数据值，而且 定义和赋值必须放到一起，不然的话就会报错，取不出来数据值，而且左边也必须是一个 js 存在数据结构不然的话也会报错，解构赋值的主要作用还是，快速的让我们在数据中抓取出我们想要的数据。

19、 let、const

let 是在定义变量，不能重复定义，但是在不同的作用域里面就可以进行再次定义，let 这种定义可以再次修改定义的数据值，let 有自己的作用域，只要是有花括号就在一个作用域，利用这个特性也可以解决循环里面加事件，事件里面的 i 值不能用的问题!let 主要在于方法和事件里面的可以修改的变量的定义，const 是定义一个常量，也是不能重复定义，只要是有一个花括号就有一个作用域，不能修改，但是这些只要是在不同的作用域中就可以

20、 async/await

Async 和 await 是一种同步的写法，但还是异步的操作，两个内容还是必须同时去写才会生效不然的话也是不会好使，而且 await 的话有一个不错的作用就是可以等到你的数据加载过来以后才会去运行下边的 js 内容，而且 await 接收的对象必须还是个 promise 对象，如果是接收数据的时候就可以直接用一句话来接收数据值，所以这个 async await 我的主要应用是在数据的接收，和异步问题的处理，主要是还是解决不同执行时机下的异步问题！

21、 es6 中函数有哪些拓展

1. 新增了块级作用域(let,const)
2. 提供了定义类的语法糖(class)
3. 新增了一种基本数据类型(Symbol)
4. 新增了变量的解构赋值
5. 函数参数允许设置默认值，引入了 rest 参数，新增了箭头函数
6. 数组新增了一些 API，如 isArray / from / of 方法;数组实例新增了 entries()，keys() 和 values() 等方法
7. 对象和数组新增了扩展运算符
8. ES6 新增了模块化(import/export)
9. ES6 新增了 Set 和 Map 数据结构
10. ES6 原生提供 Proxy 构造函数，用来生成 Proxy 实例
11. ES6 新增了生成器(Generator)和遍历器(Iterator)

22、 日常前端代码开发中，有哪些值得用 ES6 去改进的编程优化或者规范？

- 常用箭头函数来取代 `var self = this;`的做法。

- 常用 *let* 取代 *var* 命令。
- 常用数组/对象的结构赋值来命名变量，结构更清晰，语义更明确，可读性更好。
- 在长字符串多变量组合场合，用模板字符串来取代字符串累加，能取得更好地效果和阅读体验。
- 用 *Class* 类取代传统的构造函数，来生成实例化对象。
- 在大型应用开发中，要保持 *module* 模块化开发思维，分清模块之间的关系，常用 *import*、*export* 方法。

23、请写出在 vue 中使用 promise 封装项目 api 接口的思路？

axios 封装了原生的 XHR，让我们发送请求更为简单，但假设在一个成百上千个 vue 文件的项目中，我们每一个 vue 文件都要写 axios.get()或 axios.post()岂不是很麻烦？后期的维护也不方便，所以我们要对 axios 进行进一步的封装。

1.构赋 vue-cli 项目的目录如上，我们在原有的目录基础上新建 api 与 utils 文件夹，utils 里新建 request.js 文件，request.js 代码如下：

```
import axios from 'axios'
import {
  Message,
  Loading
} from 'element-ui'
import router from '../router/index.js'  //注意路径与文件名

const service = axios.create({
  baseURL: process.env.BASE_API, // api 的 base_url
  timeout: 50000 // request timeout
})

let loading // 定义loading变量

function startLoading () { // 使用Element loading-start 方法
  loading = Loading.service({
    lock: true,
    text: '加载中...',
    background: 'rgba(0, 0, 0, 0.7)'
  })
}

function endLoading () { // 使用Element loading-close 方法
  loading.close()
}
```

```
26
27  // 请求拦截  设置统一header
28  service.interceptors.request.use(
29    config => {
30      // 加载
31      startLoading()
32      if (localStorage.eleToken) {
33        config.headers.Authorization = localStorage.eleToken
34      }
35      return config
36    },
37    error => {
38      return Promise.reject(error)
39    }
40  )
41
42  // 响应拦截  401 token过期处理
43  service.interceptors.response.use(
44    response => {
45      endLoading()
46      return response
47    },
48    error => {
49      // 错误提醒
50      endLoading()
51      Message.error(error.response.data)
52
53      const { status } = error.response
54      if (status === 401) {
55        Message.error('token值无效，请重新登录')
56        // 清除token
57        localStorage.removeItem('eleToken')
```

```
58
59        // 页面跳转
60        router.push('/login')
61      }
62
63      return Promise.reject(error)
64    }
65  )
66
67  export default service
68
69
```

在 request.js 中做了三件事

1.创建 axios，设置 baseURL 与超时时间

2.拦截请求

3.拦截响应

4.路由拦截

24、 解构赋值的规则是什么以及解构赋值的用途有哪些？

规则：只要等号右边的值不是对象或数组，就先将其转为对象。

用途：

1. 交换变量的值
2. 函数参数的默认值
3. 提取 JSON 数据

25、 promise 是什么？有哪些状态和参数？如何使用？

1、主要用于异步计算

2、可以将异步操作队列化，按照期望的顺序执行，返回符合预期的结果

3、可以在对象之间传递和操作 promise，帮助我们处理队列

resolve 作用是，将 Promise 对象的状态从“未完成”变为“成功”（即从 pending 变为 resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；

reject 作用是，将 Promise 对象的状态从“未完成”变为“失败”（即从 pending 变为 rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

promise 有三个状态：

1、pending[待定]初始状态

2、fulfilled[实现]操作成功

3、rejected[被否决]操作失败

当 promise 状态发生改变，就会触发 then()里的响应函数处理后续步骤；

promise 状态一经改变，不会再变。

Promise 对象的状态改变，只有两种可能：

从 pending 变为 fulfilled

从 pending 变为 rejected。

这两种情况只要发生，状态就凝固了，不会再变了。

26、 for...in 迭代和 for...of 有什么区别

1、推荐在循环对象属性的时候，使用 for...in,在遍历数组的时候的时候使用 for...of。

2、for...in 循环出的是 key，for...of 循环出的是 value

3、注意，for...of 是 ES6 新引入的特性。修复了 ES5 引入的 for...in 的不足

4、for...of 不能循环普通的对象，需要通过和 Object.keys()搭配使用

27、 generator（异步编程、yield、next()、await 、async）

提示：Generator 是一个迭代器生成函数，其返回值是一个迭代器（Iterator），可用于异步调用。

比如某个事物只有三种状态（状态 A，状态 B，状态 C），而这三种状态的变化是状态 A =>状态 B =>状态 C =>状态 A ，这就是状态机。Generator 特别适用于处理这种状态机

28、 Ajax 是什么?以及如何创建 Ajax?

提示：

1、Ajax 并不算是一种新的技术，全称是 asychronous javascript and xml，可以说是已有技术的组合，主要用来实现客户端与服务器端的异步通信效果，实现页面的局部刷新，早期的浏览器并不能原生支持 ajax，可以使用隐藏帧（iframe）方式变相实现异步效果，后来的浏览器提供了对 ajax 的原生支持 .

2、使用 ajax 原生方式发送请求主要通过 XMLHttpRequest(标准浏览器) 、 ActiveXObject(IE 浏览器)对象实现异步通信效果 .

3、基本步骤：

```javascript
var xhr =null;//创建对象
if(window.XMLHttpRequest){
    xhr = new XMLHttpRequest();
}else{
    xhr = new ActiveXObject("Microsoft.XMLHTTP");
}
xhr.open(“方式”,”地址”,”标志位”);//初始化请求
xhr.setRequestHeader(“”,””);//设置 http 头信息
```

```
xhr.onreadystatechange =function(){}//指定回调函数
xhr.send();//发送请求
```

4、js 框架（jQuery/EXTJS 等）提供的 ajax API 对原生的 ajax 进行了封装，熟悉了基础理论，再学习别的框架就会得心应手，好多都是换汤不换药的内容

29、 同步和异步的区别?

1、同步：阻塞的

- 张三叫李四去吃饭，李四一直忙得不停，张三一直等着，直到李四忙完两个人一块去吃饭 =浏览器向服务器请求数据，服务器比较忙，浏览器一直等着（页面白屏），直到服务器返回数据，浏览器才能显示页面

2、异步：非阻塞的

- 张三叫李四去吃饭，李四在忙，张三说了一声然后自己就去吃饭了，李四忙完后自己去吃 =浏览器向服务器请求数据，服务器比较忙，浏览器可以自如的干原来的事情（显示页面），服务器返回数据的时候通知浏览器一声，浏览器把返回的数据再渲染到页面，局部更新

30、 Ajax 请求的时候 get 和 post 方式的区别?

1、理解跨域的概念：协议、域名、端口都相同才同域，否则都是跨域

2、出于安全考虑，服务器不允许 ajax 跨域获取数据，但是可以跨域获取文件内容，所以基于这一点，可以动态创建 script 标签，使用标签的 src 属性访问 js 文件的形式获取 js 脚本，并且这个 js 脚本中的内容是函数调用，该函数调用的参数是服务器返回的数据，为了获取这里的参数数据，需要事先在页面中定义回调函数，在回调函数中处理服务器返回的数据，这就是解决跨域问题的主流解决方案

31、 解释 jsonp 的原理,以及为什么 jsonp 为什么不是真正的 ajax?

答：产生跨域的情况有：不同协议，不同域名，不同端口以及域名和 ip 地址的访问都会产生跨域。

跨域的解决方案目前有三种主流解决方案：

是 jsonp

jsonp 实现原理：主要是利用动态创建 script 标签请求后端接口地址，然后传递 callback 参数，后端接收 callback，后端经过数据处理，返回 callback 函数调用的形式，callback 中的参数就是 json

优点：浏览器兼容性好，

缺点：只支持 get 请求方式

是代理（前端代理和后端通常通过 nginx 实现反向代理）

前端代理我在 vue 中主要是通过 vue 脚手架中的 config 中的 index 文件来配置的，其中有个 proxyTable 来配置跨域的

前端代理核心实现通过 http-proxy-middleware 插件来实现的，vue2.x 和 vue3.x 脚手架代理跨域实现原理是一样的

是 CORS

CORS 全称叫跨域资源共享，主要是后台工程师设置后端代码来达到前端跨域请求的

CORS 的原理：CORS 定义一种跨域访问的机制，可以让 AJAX 实现跨域访问。CORS 允许一个域上的网络应用向另一个域提交跨域 AJAX 请求。实现此功能非常简单，只需由服务器发送一个响应标头即可。

优点：无需前端工程师设置，只需后端工程师在请求的页面中配置好，并且支持所有请求方式（例如：get,post,put,delete 等）

缺点：浏览器支持有版本要求，如下：

chrome:13+,firefox:3.5+,IE 11+,edge:12+

注：现在主流框架都是用代理和 CORS 跨域实现的

32、 说一下从输入 URL 到页面加载完中间发生了什么?

答：大致过程是这样的：

1. DNS 解析
2. TCP 连接
3. 发送 HTTP 请求
4. 服务器处理请求并返回需要的数据
5. 浏览器解析渲染页面

6. 连接结束

注：这里会延伸出问 http 状态码，和三次握手和四次挥手相关问题：

（1）http 状态码：

1xx（临时响应）

表示临时响应并需要请求者继续执行操作的状态代码

2xx （成功）

表示成功处理了请求的状态码。

常见的 2 开头的状态码有：200 - 服务器成功返回网页

3xx （重定向）

表示要完成请求，需要进一步操作。 通常，这些状态代码用来重定向

常见的 3 字开头的状态码有：

301 （永久移动） 请求的网页已永久移动到新位置。 服务器返回此响应时，会自动将请求者转到新位置。

302 （临时移动） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

304 （未修改） 自从上次请求后，请求的网页未修改过。 服务器返回此响应时，不会返回网页内容。

4xx（请求错误） 这些状态代码表示请求可能出错，妨碍了服务器的处理。

常见的 4 字开头的状态有：404 - 请求的网页不存在

5xx（服务器错误）

这些状态代码表示服务器在尝试处理请求时发生内部错误。 这些错误可能是服务器本身的错误，而不是请求出错。

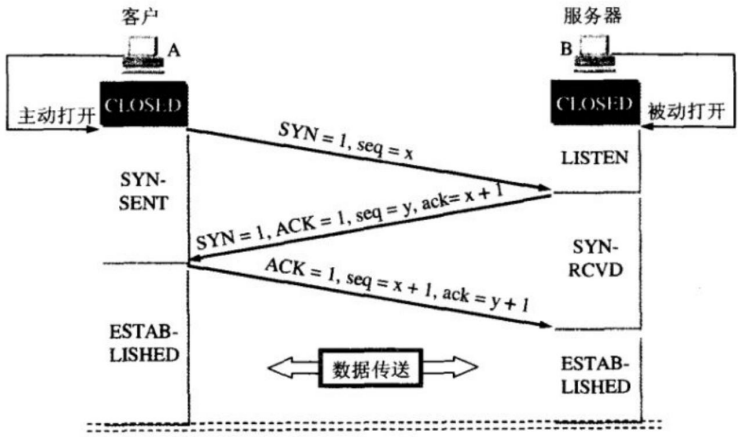
常见的以 5 开头的状态码有：

500 （服务器内部错误） 服务器遇到错误，无法完成请求。

503 （服务不可用） 服务器目前无法使用（由于超载或停机维护）。 通常，这只是暂时状态

（2）三次握手和四次挥手：

*****三次握手：**



TCP 协议中，主动发起请求的一端称为『客户端』，被动连接的一端称为『服务端』。不管是客户端还是服务端，TCP 连接建立完后都能发送和接收数据。

起初，服务器和客户端都为 CLOSED 状态。在通信开始前，双方都得创建各自的传输控制块（TCB）。
服务器创建完 TCB 后遍进入 LISTEN 状态，此时准备接收客户端发来的连接请求。

第一次握手

客户端向服务端发送连接请求报文段。该报文段的头部中 SYN=1，ACK=0，seq=x。请求发送后，客户端便进入 SYN-SENT 状态。

- PS1：SYN=1，ACK=0 表示该报文段为连接请求报文。
- PS2：x 为本次 TCP 通信的字节流的初始序号。
  TCP 规定：SYN=1 的报文段不能有数据部分，但要消耗掉一个序号。

第二次握手

服务端收到连接请求报文段后，如果同意连接，则会发送一个应答：SYN=1，ACK=1，seq=y，ack=x+1。
该应答发送完成后便进入 SYN-RCVD 状态。

- PS1：SYN=1，ACK=1 表示该报文段为连接同意的应答报文。
- PS2：seq=y 表示服务端作为发送者时，发送字节流的初始序号。
- PS3：ack=x+1 表示服务端希望下一个数据报发送序号从 x+1 开始的字节。

第三次握手

当客户端收到连接同意的应答后，还要向服务端发送一个确认报文段，表示：服务端发来的连接同意应答已经成功收到。

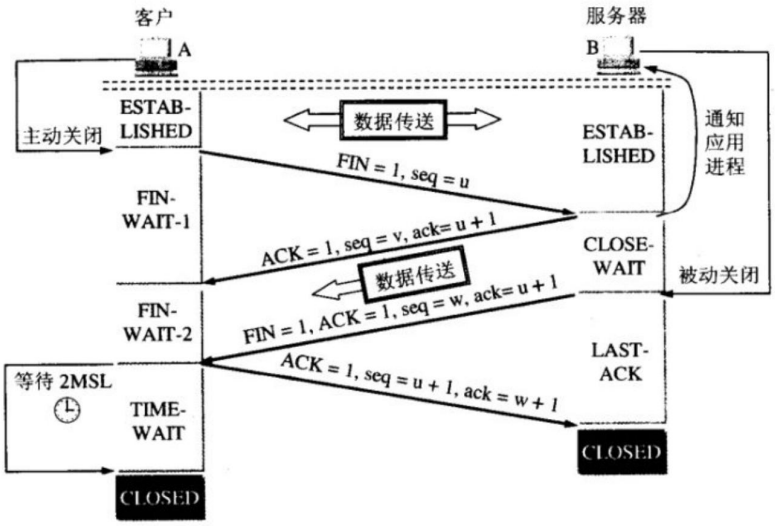
该报文段的头部为：ACK=1，seq=x+1，ack=y+1。

客户端发完这个报文段后便进入 ESTABLISHED 状态，服务端收到这个应答后也进入 ESTABLISHED 状态，此时连接的建立完成！

为什么连接建立需要三次握手，而不是两次握手？

防止失效的连接请求报文段被服务端接收，从而产生错误。

***TCP 四次挥手：



TCP 连接的释放一共需要四步，因此称为『四次挥手』。

我们知道，TCP 连接是双向的，因此在四次挥手中，前两次挥手用于断开一个方向的连接，后两次挥手用于断开另一方向的连接。

第一次挥手

若 A 认为数据发送完成，则它需要向 B 发送连接释放请求。该请求只有报文头，头中携带的主要参数为：

FIN=1，seq=u。此时，A 将进入 FIN-WAIT-1 状态。

- PS1：FIN=1 表示该报文段是一个连接释放请求。
- PS2：seq=u，u-1 是 A 向 B 发送的最后一个字节的序号。•

第二次挥手

B 收到连接释放请求后，会通知相应的应用程序，告诉它 A 向 B 这个方向的连接已经释放。此时 B 进入 CLOSE-WAIT 状态，并向 A 发送连接释放的应答，其报文头包含：

ACK=1，seq=v，ack=u+1。

- PS1：ACK=1：除 TCP 连接请求报文段以外，TCP 通信过程中所有数据报的 ACK 都为 1，表示应答。
- PS2：seq=v，v-1 是 B 向 A 发送的最后一个字节的序号。
- PS3：ack=u+1 表示希望收到从第 u+1 个字节开始的报文段，并且已经成功接收了前 u 个字节。

A 收到该应答，进入 FIN-WAIT-2 状态，等待 B 发送连接释放请求。

第二次挥手完成后，A 到 B 方向的连接已经释放，B 不会再接收数据，A 也不会再发送数据。但 B 到 A 方向的连接仍然存在，B 可以继续向 A 发送数据。

第三次挥手

当 B 向 A 发完所有数据后，向 A 发送连接释放请求，请求头：FIN=1，ACK=1，seq=w，ack=u+1。B 便进入 LAST-ACK 状态。

第四次挥手

A 收到释放请求后，向 B 发送确认应答，此时 A 进入 TIME-WAIT 状态。该状态会持续 2MSL 时间，若该时间段内没有 B 的重发请求的话，就进入 CLOSED 状态，撤销 TCB。当 B 收到确认应答后，也便进入 CLOSED 状态，撤销 TCB。

为什么 A 要先进入 TIME-WAIT 状态，等待 2MSL 时间后才进入 CLOSED 状态？

为了保证 B 能收到 A 的确认应答。

若 A 发完确认应答后直接进入 CLOSED 状态，那么如果该应答丢失，B 等待超时后就会重新发送连接释放请求，但此时 A 已经关闭了，不会作出任何响应，因此 B 永远无法正常关闭。

33、 Vue 中 methods,computed, watch 的区别

1)methods 中都是封装好的函数，无论是否有变化只要触发就会执行

适用场景：组件中功能的封装，逻辑处理

2)computed:是 vue 独有的特性计算属性，可以对 data 中的依赖项再重新计算得到一个新值，应用到视图中，和 methods 本质区别是 computed 是可缓存的，也就是说 computed 中的依赖项没有变化，则 computed 中的值就不会重新计算，而 methods 中的函数是没有缓存的。

适用场景：假设我们有一个性能开销比较大的计算属性 A，它需要遍历一个巨大的数组并做大量的计算。然后我们可能有其他的计算属性依赖于 A 。如果没有缓存，我们将不可避免的多次执行某个函数

3)Watch 是监听 data 和计算属性中的新旧变化

适用场景：当需要在数据变化时执行“异步”或“开销较大”的操作时，这个方式是最有用的

34、 prop 验证，和默认值

所有的 prop 都使得其父子 prop 之间形成了一个单向下行绑定：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解。

额外的，每次父级组件发生更新时，子组件中所有的 prop 都将会刷新为最新的值。这意味着你不应该在一个子组件内部改变 prop。如果你这样做了，Vue 会在浏览器的控制台中发出警告。子组件想修改时，只能通过 $emit 派发一个自定义事件，父组件接收到后，由父组件修改。

有两种常见的试图改变一个 prop 的情形 ：

(1)**这个用来传递一个初始值；这个子组件接下来希望将其作为一个本地的数据来使用。** 在这种情况下，最好定义一个本地的属性并将这个用作其初始值：

```
props: ['initialCounter'],
data: function () {
  return {
    counter: this.initialCounter
  }
}
```

(2)**这个以一种原始的值传入且需要进行转换。** 在这种情况下，最好使用这个的值来定义一个计算属性

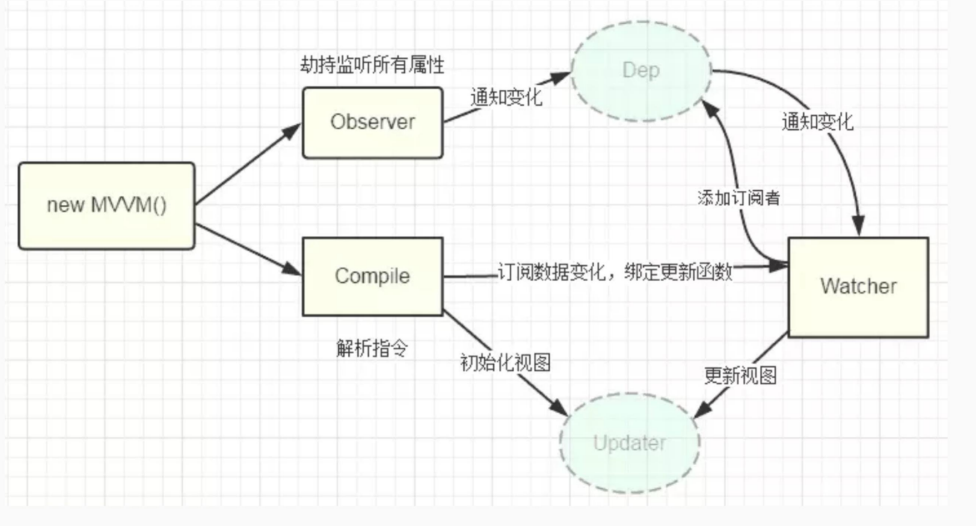
```
props: ['size'],
computed: {
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```

35、 vue 双向数据绑定原理

1、实现一个数据监听器 Observer，能够对数据对象的所有属性进行监听，如有变动可拿到最新值并通知订阅者

2、实现一个指令解析器 Compile，对每个元素节点的指令进行扫描和解析，根据指令模板替换数据，以及绑定相应的更新函数

3、实现一个 Watcher，作为连接 Observer 和 Compile 的桥梁，能够订阅并收到每个属性变动的通知，执行指令绑定的相应回调函数，从而更新视图



36、 vue 组件父子，子父，兄弟通信

第一种：父传子：主要通过 props 来实现的

具体实现：父组件通过 import 引入子组件，并注册，在子组件标签上添加要传递的属性，子组件通过 props 接收，接收有两种形式一是通过数组形式[‘要接收的属性’]，二是通过对象形式{ }来接收，对象形式可以设置要传递的数据类型和默认值，而数组只是简单的接收

第二种：子传父：主要通过$emit 来实现

具体实现：子组件通过通过绑定事件触发函数，在其中设置 this.$emit(‘要派发的自定义事件’，要传递的值)，$emit 中有两个参数一是要派发的自定义事件，第二个参数是要传递的值

第三种：兄弟之间传值有两种方法：

方法一：通过 event bus 实现

具体实现:创建一个空的 vue 并暴露出去，这个作为公共的 bus,即当作两个组件的桥梁，在两个兄弟组件中分别引入刚才创建的 bus，在组件 A 中通过 bus.$emit（’自定义事件名’，要发送的值）发送数据，在组件 B 中通过 bus.$on（‘自定义事件名‘,function(v) { //v 即为要接收的值 }）接收数据

方法二：通过 vuex 实现

具体实现：vuex 是一个状态管理工具，主要解决大中型复杂项目的数据共享问题，主要包括 state,actions,mutations,getters 和 modules 5 个要素，主要流程：组件通过 dispatch 到 actions，actions 是异步操作，再 actions 中通过 commit 到 mutations，mutations 再通过逻辑操作改变 state，从而同步到组件，更新其数据状态

37、 vue 生命周期

beforeCreate(创建前)

在实例初始化之后，数据观测和事件配置之前被调用，此时组件的选项对象还未创建，el 和 data 并未初始化，因此无法访问 methods， data， computed 等上的方法和数据。

created (创建后）

实例已经创建完成之后被调用，在这一步，实例已完成以下配置：数据观测、属性和方法的运算，watch/event 事件回调，完成了 data 数据的初始化，el 没有。然而，挂在阶段还没有开始, $el 属性目前不可见，这是一个常用的生命周期，因为你可以调用 methods 中的方法，改变 data 中的数据，并且修改可以通过 vue 的响应式绑定体现在页面上，，获取 computed 中的计算属性等等，通常我们可以在这里对实例进行预处理，也有一些童鞋喜欢在这里发 ajax 请求，值得注意的是，这个周期中是没有什么方法来对实例化过程进行拦截的，因此假如有某些数据必须获取才允许进入页面的话，并不适合在这个方法发请求，建议在组件路由钩子 beforeRouteEnter 中完成

beforeMount

挂在开始之前被调用，相关的 render 函数首次被调用（虚拟 DOM），实例已完成以下的配置：编译模板，把 data 里面的数据和模板生成 html，完成了 el 和 data 初始化，注意此时还没有挂在 html 到页面上。

mounted

挂在完成，也就是模板中的 HTML 渲染到 HTML 页面中，此时一般可以做一些 ajax 操作，mounted 只会执行一次。

beforeUpdate

在数据更新之前被调用，发生在虚拟 DOM 重新渲染和打补丁之前，可以在该钩子中进一步地更改状态，不会触发附加地重渲染过程

updated（更新后）

在由于数据更改导致地虚拟 DOM 重新渲染和打补丁只会调用，调用时，组件 DOM 已经更新，所以可以执行依赖于 DOM 的操作，然后在大多是情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环，该钩子在服务器端渲染期间不被调用

beforeDestrioy （销毁前）

在实例销毁之前调用，实例仍然完全可用，

这一步还可以用 this 来获取实例，

一般在这一步做一些重置的操作，比如清除掉组件中的定时器和监听的 dom 事件

destroyed（销毁后）

在实例销毁之后调用，调用后，所以的事件监听器会被移出，所有的子实例也会被销毁，该钩子在服务器端渲染期间不被调用

38、 说一下 vue 开发环境和线上环境如何切换

答：主要通过检测 process.env.NODE_ENV===”production”和 process.env.NODE_ENV===”development”环境，来设置线上和线下环境地址，从而实现线上和线下环境地址的切换，通常在封装 http 请求时，抽取出环境变量所依赖的环境（生产，开发，测试）地址，在封装好的 http 请求中引入，从而方便切换环境地址。

39、 vue 路由传参数如何实现

ps:主要通过 query 和 params 来实现

query 传参：通过在 router-link 或 this.$router.push()传递 url 地址并且拼接?问号传递的参数 例如：rourer-link to=”/goods?id=1001”,然后在接收的页面通过 this.$route.query.id 来接收

优点：通用性比较好，刷新数据不会丢失

params 传参：通过在 router-link 或 this.$router.push()传递 url 地址并且拼接的参数也用/来表示 例如：rourer-link to=”/goods/1001”,并且在路由页面通过 routes=[{path:’/goods/:id’}]配置，最后在接收的页面通过 this.$route.params.id 来接收

优点：传递数据量在，优雅

缺点：刷新数据不会丢失

40、 路由导航守卫有几种，如何实现

一、全局路由守卫

所谓全局路由守卫，就是小区大门，整个小区就这一个大门，你想要进入其中任何一个房子，都需要经过这个大门的检查

全局路由守卫有个两个：一个是全局前置守卫，一个是全局后置守卫

router.beforeEach((to, from, next) => {

console.log(to) => // 到哪个页面去？

console.log(from) => // 从哪个页面来？

next() => // 一个回调函数

}

router.afterEach（to，from） = {}

next()：回调函数参数配置

next(false): 中断当前的导航。如果浏览器的 URL 改变了 (可能是用户手动或者浏览器后退按钮)，那么 URL 地址会重置到 from 路由对应的地址

next('/') 或者 next({ path: '/' }): 跳转到一个不同的地址。当前的导航被中断，然后进行一个新的导航。你可以向 next 传递任意位置对象，且允许设置诸如 replace: true、name: 'home' 之类的选项以及任何用在 router-link 的 to prop 或 router.push 中的选项

二、组件路由守卫

// 跟 methods: {}等同级别书写，组件路由守卫是写在每个单独的 vue 文件里面的路由守卫

beforeRouteEnter (to, from, next) {

// 注意，在路由进入之前，组件实例还未渲染，所以无法获取 this 实例，只能通过 vm 来访问组件实例

next(vm => {})

}

beforeRouteUpdate (to, from, next) {

// 同一页面，刷新不同数据时调用，

}

beforeRouteLeave (to, from, next) {

// 离开当前路由页面时调用

}

三、路由独享守卫

路由独享守卫是在路由配置页面单独给路由配置的一个守卫

export default new VueRouter({

routes: [

```
        {
            path: '/',
            name: 'home',
            component: 'Home',
            beforeEnter: (to, from, next) => {
                // ...
            }
        }
    ]
})
```

41、 vuex 中 state,getters,mutations,actions,modules,plugins 的用途，和用法

state：存储状态（变量）

getters：对数据获取之前的再次编译，可以理解为 state 的计算属性。我们在组件中使用 $sotre.getters.fun()

mutations：修改状态，并且是同步的。在组件中使用 $store.commit('',params)。这个和我们组件中的自定义事件类似。

actions：异步操作。在组件中使用是$store.dispath('')

modules：store 的子模块，为了开发大型项目，方便状态管理而使用的。这里我们就不解释了，用起来和上面的一样。

42、 vue 中 key 的作用

“key 值:用于管理可复用的元素。因为 Vue 会尽可能高效地渲染元素,通常会复用已有元素而不是从头开始渲染。这么做使 Vue 变得非常快,但是这样也不总是符合实际需求。 2.2.0+ 的版本里,当在组件中使用 v-for 时,key 是必须的。”

43、 vue 自定义指令如何使用

【全局指令】

使用 Vue.diretive()来全局注册指令。

【局部指令】

也可以注册局部指令，组件或 Vue 构造函数中接受一个 directives 的选项。

钩子函数。指令定义函数提供了几个钩子函数（可选）。

【bind】

只调用一次，指令第一次绑定到元素时调用，用这个钩子函数可以定义一个在绑定时执行一次的初始化动作。

【inserted】

被绑定元素插入父节点时调用（父节点存在即可调用，不必存在于 document 中）。

【update】

所在组件的 VNode 更新时调用，但是可能发生在其孩子的 VNode 更新之前。指令的值可能发生了改变也可能没有。但是可以通过比较更新前后的值来忽略不必要的模板更新。

【componentUpdated】

所在组件的 VNode 及其孩子的 VNode 全部更新时调用。

【unbind】

只调用一次，指令与元素解绑时调用。

钩子函数参数

钩子函数被赋予了以下参数

【el】

指令所绑定的元素，可以用来直接操作 DOM。

【binding】

一个对象，包含以下属性：

name: 指令名，不包括 v- 前缀。

value: 指令的绑定值，例如： v-my-directive="1 + 1", value 的值是 2。

oldValue: 指令绑定的前一个值，仅在 update 和 componentUpdated 钩子中可用。无论值是否改变都可用。

expression: 绑定值的字符串形式。例如 v-my-directive="1 + 1" ， expression 的值是 "1 + 1"。

arg: 传给指令的参数。例如 v-my-directive:foo， arg 的值是 "foo"。

modifiers: 一个包含修饰符的对象。例如： v-my-directive.foo.bar, 修饰符对象 modifiers 的值是 { foo: true, bar: true }。

44、 vue 常用修饰符

.stop: 阻止点击事件冒泡。等同于 JavaScript 中的 event.stopPropagation()

使用了.stop 后，点击子节点不会捕获到父节点的事件

.prevent 防止执行预设的行为（如果事件可取消，则取消该事件，而不停止事件的进一步传播），等同于 JavaScript 中的 event.preventDefault()，prevent 等同于 JavaScript 的 event.preventDefault()，用于取消默认事件。

.capture 与事件冒泡的方向相反，事件捕获由外到内,捕获事件：嵌套两三层父子关系，然后所有都有点击事件，点击子节点，就会触发从外至内父节点-》子节点的点击事件

.self 只会触发自己范围内的事件，不包含子元素

.once 只执行一次，如果我们在@click 事件上添加.once 修饰符，只要点击按钮只会执行一次。

.passive Vue 还对应 addEventListener 中的 passive 选项提供了 .passive 修饰符

45、 keep-alive 的作用

1、 在 vue 项目中,难免会有列表页面或者搜索结果列表页面,点击某个结果之后,返回回来时,如果不对结果页面进行缓存,那么返回列表页面的时候会回到初始状态,但是我们想要的结果是返回时这个页面还是之前搜索的结果列表,这时候就需要用到 vue 的 keep-alive 技术了.

2、 在 router-view 上使用可以缓存该路由组件

3、 有两个参数 include - 字符串或正则表达，只有匹配的组件会被缓存

exclude - 字符串或正则表达式，任何匹配的组件都不会被缓存

46、 vue 路由有几种模式

对于 Vue 这类渐进式前端开发框架，为了构建 SPA（单页面应用），需要引入前端路由系统，这也就是 Vue-router 存在的意义。前端路由的核心，就在于————改变视图的同时不会向后端发出请求。

为了达到这个目的，浏览器提供了以下两种支持：

1、hash ——即地址栏 URL 中的#符号（此 hsah 不是密码学里的散列运算）。比如这个 URL：http://www.abc.com/#/hello, hash 的值为#/hello。它的特点在于：hash 虽然出现 URL 中，但不会被包含在 HTTP 请求中，对后端完全没有影响，因此改变 hash 不会重新加载页面。

2、history ——利用了 HTML5 History Interface 中新增的 pushState() 和 replaceState() 方法。（需要特定浏览器支持）

47、 Object.defineProperty()方法有何作用

```
var o={};//创建一个新对象
Object.defineProperty(o,"a",{
    //该属性对应的值。可以是任何有效的 JavaScript 值（数值，对象，函数等）。默认为 undefined
    value:37,
    // 仅当仅当该属性的writable为 true 时，该属性才能被赋值运算符改变。默认为 false
    writable:true,
    //仅当该属性的 enumerable 为 true 时，该属性才能够出现在对象的枚举属性中。默认为 false
    enumerable:true,
    // 仅当该属性的 configurable 为 true 时，该属性才能够被改变，也能够被删除。默认为 false
    configurable:true
});
// 对象o拥有了属性a，值为37

var bValue;
Object.defineProperty(o,"b",{
    get:function () { return bValue; },
    set:function (v) { bValue=v; },
    enumerable:true,
    configurable:true
});
o.b=45;
```

1、语法 Object.defineProperty(obj, prop, descriptor)

2、定义：Object.defineProperty() 直接在一个对象上定义一个新属性，或者修改现有属性，并返回该对象。

3、参数

obj 要在其上定义属性的对象。

prop 要定义或修改的属性的名称。

descriptor 将被定义或修改的属性描述符。

4、返回值

返回被操作的对象，即返回 obj 参数

5、注意点

1）当把 configurable 值设置为 false 后,就不能修改任何属性了,包括自己本身这个属性

2）想用访问器属性模拟默认行为的话,必须得在里面新顶一个属性,不然的话会造成循环引用

3）可枚举属性对 for/in, Object.keys(), JSON.stringify(), Object.assign() 方法才生效(for/in 是对所有可枚举属性，而其他三种是对自身可枚举属性)

6、用途

1）vue 通过 getter-setter 函数来实现双向绑定

2）俗称属性挂载器

3）专门监听对象数组变化的 Object.observe()(es7)也用到了该方法

48、什么是虚拟 dom，和 diff 算法

1、虚拟 DOM 的最终目标是将虚拟节点渲染到视图上。但是如果直接使用虚拟节点覆盖旧节点的话，会有很多不必要的 DOM 操作。例如，一个 ul 标签下很多个 li 标签，其中只有一个 li 有变化，这种情况下如果使用新的 ul 去替代旧的 ul,因为这些不必要的 DOM 操作而造成了性能上的浪费。

为了避免不必要的 DOM 操作，虚拟 DOM 在虚拟节点映射到视图的过程中，将虚拟节点与上一次渲染视图所使用的旧虚拟节点（oldVnode）做对比，找出真正需要更新的节点来进行 DOM 操作，从而避免操作其他无需改动的 DOM。

简而言之主要做了两件事：

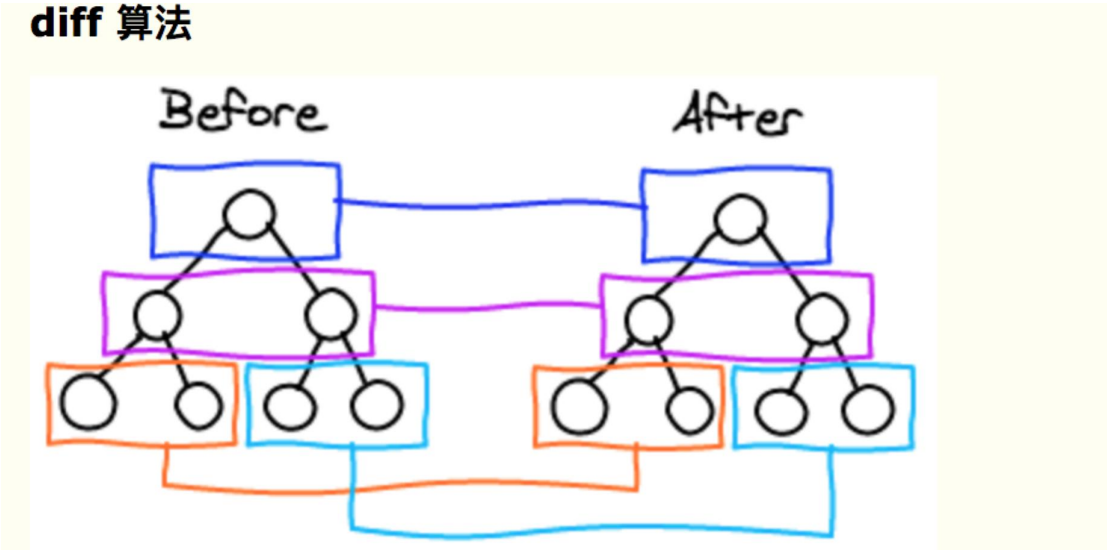
提供与真实 DOM 节点所对应的虚拟节点 vnode 将虚拟节点 vnode 和旧虚拟节点 oldVnode 进行对比，然后更新视图

2、diff 算法包括几个步骤：

a.用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中

b.当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异

c.把所记录的差异应用到所构建的真正的 DOM 树上，视图就更新了



49、 vue 中数组中的某个对象的属性发生变化，视图不更新如何解决

问题原因：因为 vue 的检查机制在进行视图更新时无法监测 数组中的某个对象的属性值的变化。解决方案如下

方案一：利用 this.set(this.obj,key,val)

例：this.set(this.obj,‘k1’,‘v1’)

方案二：就利用 Object.assign({}，this.obj)创建新对象

如果是数组就 Object.assign([]，this.obj)

如果是对象就 Object.assign({}，this.obj)

50、 vue3.0 与 vue2.0 的区别

1、性能提升

一句话简介：更小巧，更快速；支持摇树优化；支持 Fragments 和跨组件渲染；支持自定义渲染器。

2、API 变动

一句话介绍：除渲染函数 API 和 scoped-slot 语法之外，其余均保持不变或者将通过另外构建一个兼容包 来兼容 2.x。

模板语法的 99% 将保持不变。除了 scoped slot 语法可能会有一些微调之外变动最大的部分将是渲染函数 (render) 中的虚拟 DOM 的格式。

3、重写虚拟 DOM (Virtual DOM Rewrite)

随着虚拟 DOM 重写，减少 运行时（runtime）开销。重写将包括更有效的代码来创建虚拟节点。